Output-Sensitive Parallel Algorithm for Polygon Clipping

Satish Puri and Sushil K. Prasad
Department of Computer Science
Georgia State University
Atlanta - 30303, USA
Email: spuri2@student.gsu.edu, sprasad@gsu.edu

Abstract—Polygon clipping is one of the complex operations in computational geometry. It is a primitive operation in many fields such as Geographic Information Systems (GIS), Computer Graphics and VLSI CAD. Sequential algorithms for this problem are in abundance in literature but there are very few parallel algorithms solving it in its most general form. We present the first output-sensitive CREW PRAM algorithm, which can perform polygon clipping in $O(\log n)$ time using $(n + k + k')$ processors, where $n$ is the number of vertices, $k$ is the number of edge intersections and $k'$ is the additional temporary vertices introduced due to the partitioning of polygons. The current best algorithm by Karinthi, Srinivas, and Almasi [1] does not handle self-intersecting polygons, is not output-sensitive and must employ $\Theta(n^2)$ processors to achieve $O(\log n)$ time. Our algorithm is developed from the first principles and it is superior to [1] in cost. It yields a practical implementation on multicore and demonstrates 30x speedup for real-world dataset. Our algorithm can perform the typical clipping operations including intersection, union, and difference.

I. INTRODUCTION

In geometry, a polygon is traditionally a plane figure that is bounded by a finite chain of straight line segments closing in a loop to form a closed chain. A polygon can be classified as a simple polygon in case its line segments do not intersect among themselves, otherwise it is termed as self-intersecting. A polygon is convex if every interior angle is less than or equal to 180 degrees otherwise it is concave. Geometric intersection problems arise naturally in a number of applications. Examples include clipping in graphics, wire and component layout in VLSI and map overlay in geographic information systems (GIS). Clipping polygons is a fundamental operation in image synthesis and it is widely used in Computer Graphics. While clipping usually involves finding the intersections (regions of overlap) of subject and clip polygons, some clipping algorithms can also find union and difference. Clipping an arbitrary polygon against an arbitrary polygon is a complex task. The major computation in clipping involves 1) finding edge intersections, 2) classifying input vertices as contributing or non-contributing to output, and 3) stitching together the vertices generated in Step 1 and 2 to produce one or more output polygons.

The polygonal data has high degree of irregularity. For two polygons with $n$ and $m$ edges, the number of intersections can be $O(nm)$. Plane-sweep based polygon clipping algorithm has time complexity of $(n + k)\log n$ where $k$ is the number of intersections [2]. Convex polygons are easier to handle in comparison to non-convex and self-intersecting polygons. With polygons that are not self-intersecting, it suffices to test edges taken from two different polygons for intersection detection. On the other hand, self-intersecting polygons increases the number of edge pairs for intersection detection since now an edge from a polygon can intersect with other edges from the same polygon. In general, the complexity of clipping operation between two polygons depends on 1) number of input polygon vertices, 2) number of intersections, and 3) self-intersection in edges.

GIS polygonal datasets may contain large number of arbitrary polygons. Some of the U.S. State boundaries consists of about 100,000 polygonal edges. Parallel polygon clipping algorithms in these scenarios can improve the performance when compared with sequential clipping algorithms. In this paper, we present the parallelization of Vatti’s polygon clipping algorithm [3] using PRAM model. Our major objective is to come up with parallel clipping algorithm design for PRAM model that is output-sensitive and run as fast as theoretically possible.

Our specific technical contributions are as follows:

1) We present parallelization of a plane-sweep based Vatti’s algorithm relying only on primitives such as prefix sum and sorting, and handling arbitrary polygons. The insights and algorithmic tools developed for the plane-sweep based algorithm may also be useful elsewhere.

2) Our algorithm is the first output-sensitive polygon clipping algorithm with $O(((n + k + k')\log(n + k + k'))/p)$ time complexity using $p$ processors, where $k' \leq O(n^2)$ is the additional temporary vertices introduced due to the partitioning of polygons and $p \leq O(n + k + k')$. This improves the current state of art which is not output-sensitive, always employs $\theta(n^2)$ processors and does not handle self-intersecting polygons [1].

We present a new technique to parallelize plane-sweep based on 1) counting the pairs of inversions and 2) reporting them. Our technique can be easily implemented by parallel sorting as opposed to some of the existing
techniques that use complex data structures such as array-of-trees or plane-sweep trees [4], [5] to parallelize plane-sweep.

3) We also present a practical, multi-threaded version of the plane-sweep based algorithm and its implementation for clipping of arbitrary polygons, and validate its performance using real-world and simulated datasets. We obtained 30x speedup using a 64-core 1.4 GHz AMD Opteron Processor for real-world dataset containing large number of polygons when compared to sequential clipping libraries and ArcGIS which is the fastest currently available commercial software application that we found. For baseline performance, we used ArcGIS version 10 running on 64-bit Windows 7 with AMD Phenom Processor 2.7 GHz with 8 GB memory.

The rest of this paper is organized as follows: Section II reviews the relevant literature briefly, introduces various operations that define polygon clipping, and reviews the segment tree [6] data structure utilized in our algorithm. Section III describes our plane-sweep based polygon clipping algorithm, with rigorous design and time complexity analysis. Our multi-threaded algorithm is in Section IV and its experimental results are in Section V. Section VI concludes the paper.

II. BACKGROUND AND LITERATURE

A. The Computational Model

The PRAM (parallel random access machine) model of parallel computation is a fundamental shared-memory model where the processors operate synchronously. The CREW (concurrent read exclusive write) version of this model allows many processors to simultaneously read the content of a memory location, but forbids any two processors from simultaneously attempting to write in the same memory location. Since we wish to solve polygon clipping in an output-sensitive manner, we allow the number of processors to grow based on the output size. As such, additional processors are requested (if necessary) constant number of times. This previously-established approach has been employed, for example, in [4], [7], and [8], for line segment intersection problems. We note that [8] also employed dynamic spawning, which we do not need.

B. Polygon clipping

Sutherland-Hodgman [9] and Liang-Barsky clipping [10] algorithms are sequential clipping algorithms which do not work for arbitrary polygons and the existing parallel clipping algorithms we found in literature are parallelization of these sequential algorithms with implementation done on classic parallel architectures [11]–[14]. Vatti’s algorithm is sequential polygon clipping algorithm which takes arbitrary polygons as input. To our best knowledge, there is no parallelization of this algorithm in literature. Apart from Vatti’s algorithm, plane-sweep based sequential polygon intersection algorithms are also discussed in [2], [15], [16]. Parallelization of plane-sweep algorithm for multi-cores is discussed in [17], [18]. However, our parallelization focuses on polygon clipping specifically and our partitioning and merging techniques are also different from [17]. The algorithm described in [18] is applicable for rectangles.

Limitations of Current Parallel Algorithms: There are only a few papers on computing set operations on polygons in parallel [1], [19] (even though there are several parallel algorithms for line intersections [4], [7], [8]). Self-intersections are not handled in [1]. The time complexity of algorithm described in [1] is using \( O(n \log n) \) processors on CREW PRAM model, thus is not sensitive to output size. Other solutions for parallel polygon clipping [11]–[14] are limited to certain types of polygons and not output sensitive. A uniform grid based partitioning approach is discussed in [19] where a grid is superimposed over the input polygons. Intersection finding operation and output vertex processing is performed in parallel in the grid cells. Thus, this works well only with good load distribution. A survey on parallel geometric intersection problems is presented in [20]. In contrast, our parallel algorithm is output-sensitive since our algorithm does not need to know the number of edges in the output polygon(s) in advance. It also does not need to know the number of processors required to optimally solve the problem in advance. Moreover, it can handle arbitrary polygons.

C. Segment Tree

We employ segment tree data structure. It is a complete binary tree as shown in Figure 1. Its leaves correspond to the elementary intervals induced by the endpoints of segments and its internal nodes correspond to intervals that are the union of elementary intervals. The internal nodes contain list of segments (cover list) which span the node’s range (i.e., the node range is a subinterval of the horizontal projection of these segments), but do not span the range of node’s parent. A query for a segment tree, receives a point \( p \), and retrieves a list of all the segments stored which contain \( p \) in time where \( k \) is the number of retrieved segments. A segment tree with cover lists can be constructed in time using \( O(n) \) processors on CREW PRAM model [21]. Variations of segment tree known as plane-sweep tree has been explored in literature to parallelize plane-sweep algorithm for line-segment intersection problem [4], [8].
III. ALGORITHM FOR CLIPPING TWO POLYGONS

A. Terminology

We present the clipping algorithm for two input polygons namely subject polygon and clip polygon. This algorithm can be extended to handle two sets of input polygons. A polygon can be viewed as a set of left and right bounds with respect to the interior of the polygon as described in [3]. A bound comprises of edges starting at a local minima and ending at a local maxima. All the edges on the left bound are called left edges and those on the right are called right edges. Similarly, vertices can be labeled as minima, maxima, left and right [3]. When two polygons are overlaid to produce the output polygon, some edges/vertices will be part of the output and we call them as contributing edges/vertices. An imaginary horizontal line called scanline (also known as sweepline) is drawn through each vertex such that there are no vertices in between two such lines. The area between two successive scanlines form a scanbeam.

B. Vatti’s Polygon Clipping Algorithm

Vatti’s algorithm uses a variation of plane-sweep method to clip a pair of polygons or two sets of polygons. Start and end vertices of edges are treated as event points. The algorithm creates minima table to store the minima list and bounds of subject and clip polygon. Also, a scanbeam table is created to store the event point schedule. An active edge list E containing all the edges passing through a scanbeam is maintained and updated dynamically as the plane-sweep progresses from one scanbeam to another. The gist of Vatti’s algorithm is to scan every vertex in the input polygons from bottom to top, and compute intersection points in E. The output polygon is constructed incrementally by stitching the contributing vertices and intersections together based on their labels.

The intersection of two polygons with self-intersecting edges is shown in Figure 2. Let us consider scanbeam s2–s6. The active edges passing through this scanbeam are s7s6, c9c8, c4c5 and s2s3. At first, intersecting vertices I5, I6 and I7 are found out and processed. When two polygonal edges intersect to generate a vertex, then the nature of the edge intersection can determine the relative position of the vertex in the output polygon. The vertex labeling rules are discussed in Vatti’s algorithm [3]. For example, I6 is the intersection between edges c4c5 and s7s6, both with label left, I6 is tagged with left label. Applying the vertex labeling rules, I2 is tagged with maxima label and I3 is tagged with minima label. A minima signifies initialization of new polygon and a maxima signifies termination of a polygon. Moreover, these labels also determine the relative position of the vertices in the output polygon. We use these insights in our parallel algorithm.

In subsection III-E we show how active edges can be independently determined in the scanbeams by concurrently querying a parallel segment tree. Once the active edges are determined, these edges can be labeled independently as shown in Lemma 1 and contributing vertices can be identified independently as shown in Lemma 2 in all the scanbeams.

C. Problem Definition

Let polygon \( B = (V_b, E_b) \) where \( V_b = \{b_1, b_2, \ldots, b_n\} \), \( b_i \in B \) represent \( n \) vertices and \( E_b = \{b_1b_2, b_2b_3, \ldots, b_{n-1}b_n, b_nb_1\} \) represent the edges. Let polygon \( O = (V_o, E_o) \) where \( V_o = \{o_1, o_2, \ldots, o_t\}, o_i \in O \) represent \( t \) vertices and \( E_o = \{o_1o_2, o_2o_3, \ldots, o_{t-1}o_t, o_to_1\} \) represent the edges. Each vertex is represented by a pair of cartesian coordinates \( x \) and \( y \). Let \( I \) represent vertices of intersection among edges in \( B \) and \( O \). For simplicity, we assume that there are no horizontal edges (parallel to x-axis) present. If horizontal edges are present then we assume that the edges are preprocessed by slightly perturbing the vertices to make them non-horizontal. Given, \( B, O \) and an operator \( \text{op} \), we want to find the set \( P \), which represents the new polygon(s) consisting of vertices from \( V_o \), \( O_b \) and \( I \) and edges from \( E_b, E_o \) and new edges induced by \( I \). The set \( P \) may represent zero or more polygons (zero in case when input polygons do not overlap). Clipping may produce a point or a line. For simplicity, we assume no such cases. We concentrate on output which can be a convex polygon or a concave polygon or a set of convex and/or concave polygons.

D. Multi-Way Divide and Conquer Algorithm

Overview of plane-sweep based algorithm: Let Y = \( \{y_1, y_2, \ldots, y_m\} \) be an ordered set of vertices formed by projecting the edge start and end vertices on y-axis. Each of the \( m-1 \) scanbeams from bottom to top are associated with intervals \( \{(y_1, y_2), (y_2, y_3), \ldots, (y_{m-1}, y_m)\} \). The intervals \( \{y_i, y_{i+1}\} \) with \( y_i \) equal to \( y_{i+1} \) are not considered as they do not form a valid scanbeam.

The edges in polygons \( B \) and \( O \) are merged together and partitioned into multiple scanbeams. Partitioning the input polygons by scanlines decomposes the original problem into subproblems (of various sizes) represented by the scanbeams, suitable to apply divide and conquer approach. New geometrical figures (partial polygons) are formed by intersecting the m
scanlines with $B$ and likewise with $O$. The additional vertices thus produced by intersecting scanlines with input polygons are denoted as virtual vertices and these are stored along with individual partial polygons as pivots which can be used later in the merging phase. Figure 3 shows examples of partial polygons that are possible in a scanbeam. Also note in this figure, how the left($L$) and right($R$) edge labels alternate one another in a scanbeam. In Lemma 1, we provide a proof sketch for this property.

The following results are on concurrent processing of vertices and edges in a scanbeam and its time complexity.

**Lemma 1: Labeling Locally:** The polygon edges in a scanbeam can be labeled as left or right without looking at the overall geometry and the labels alternate one another (Figure 3).

*Proofsketch:* Based on the property of winding number, it is shown that an imaginary ray that intersects the polygon exactly once traverses either from the interior to the exterior of the polygon, or vice versa. Based on this fact, we can observe that if a point moves along a ray from infinity to the point and if it crosses the boundary of a polygon then it alternately goes from the outside to inside, then from the inside to the outside, etc. Since, the edge labeling (left or right) is defined in terms of interior of a polygon, if edges are arranged in the order in which they intersect the scanline, the edges of a polygon can be labeled as left and right alternately.

Thus, from Lemma 1, we can derive an algorithm to find the label of an edge based on its index in the sorted edge list in a scanbeam. An intersection vertex can be labeled as *Left, Right, Minima* or *Maxima* based on the labels of the intersecting edges using Vatti’s vertex labeling rules. Thus, as a consequence of Lemma 1, relative position of a vertex can also be independently determined in the partial output polygon.

**Lemma 2: Independent Identification of Contributing Vertices:** A vertex can be classified as contributing or not independently in a scanbeam.

*Proofsketch:* In a scanbeam, a vertex can either be a start/end vertex or a new intersection. As such, two cases are possible:

Case 1: Intersection vertex: The vertices lying in between the scanlines are intersecting points which can be either i) intersections between the edges from the same polygon or ii) from different polygons. In case ii), the vertices are contributing irrespective of the clipping operation involved.

Case i) depends on the clipping operation involved and it can be resolved using point-in-partial-polygon test using only the edges in a scanbeam which we explain shortly.

*Proofsketch:* In a sequential point-in-polygon testing, one pass through all the edges of a polygon is required in order to determine if the point is contributing or not. But, when scanbeams are already created, this test can be performed faster since only the edges in a given scanbeam needs to be considered, other edges can be safely ignored. Assuming that the edges in a scanbeam denoted by $E$ are sorted (by $x$-coordinate of edge’s intersection with scanline), in order to find if a vertex from subject polygon is contributing or not, the number of edges from clip polygon to its left in $E$ are counted. If the count is odd then the edge is contributing otherwise non-contributing. This parity test can be expressed as a standard prefix sum problem.

**Lemma 3: Contributing Vertices in $O(\log n)$ Time:** On PRAM model, a logarithmic time algorithm can be developed to find whether a vertex on a scanline is contributing or not.

*Proofsketch:* Let us consider a set $E$ of $n$ edges in a scanbeam which is sorted on the $x$-coordinate of intersection of $E$ with a given scanline. For this ordered set, let $L = [l_1, l_2, ..., l_n]$ represent labels for the corresponding edges in $E$ and a binary associative operator +. Assign 0 to the label of the edges belonging to $B$ and 1 to those belonging to $O$. A prefix sum for $L$ is denoted by $P[l_1, (l_1 + l_2), ..., (l_1 + l_2 + .. + l_n)]$, where $P_i$ is represented as $(l_1 + l_2 + .. + l_i)$. A vertex $v$ of an edge $E_i \in B$ lying on a scanline, is contributing if and only if $P_i$ is odd. Now, repeat the prefix sum by reversing the 0/1 label for the edges of $B$ and $O$ to determine the contributing vertices for polygon $O$. This all-prefix-sum even-odd parity test can find all the contributing vertices in a scanbeam by invoking it twice (for lower and upper scanline). This algorithm requires sorting and prefix sum computation. With $n$ processors, both operations can be done in $O(\log n)$ time.

**Finding Intersections using Inversions:** If the edges span a bounded region, number of edge intersections can be found out within the region simply by knowing the order in which the edges intersect the boundary of the region.

Fig. 3: 1) Different types of partial polygons in a scanbeam with edge labels II) Assigning side label to an edge (L stands for left, R stands for right)

Fig. 4: Intersection of 4 edges in a scanbeam. The inversion pairs are (3,1), (3,2), (4,1), (2,1) for the list \{3,2,4,1\}. These pairs also represent the intersecting edges.

---

**Finding Intersections using Inversions:** If the edges span a bounded region, number of edge intersections can be found out within the region simply by knowing the order in which the edges intersect the boundary of the region [25]. For example,
as shown in Figure 4, the order of edges L intersecting the lower scanline is \{3, 2, 4, 1\} and the number of inversions in L is equal to the number of edge intersections in the scanbeam. In the worst case, there can be \(O(n^2)\) inversions in an unsorted list but by extending mergesort, inversions in a scanbeam can be counted in \(O(n \log n)\) time sequentially. If we add the inversions for all the scanbeams, the number of intersections between two arbitrary polygons can be found out. To allocate processes in an output-sensitive manner, we need to first find the number of intersections and then allocate that number of additional processors to the pairs of intersecting segments.

**Lemma 4:** The number of intersections in a scanbeam can be computed in \(O(\log n)\) time using \(O(n)\) processors.

**Proof sketch:** We show how parallel mergesort can be extended to find the intersecting pairs of edges by counting the number of inversions first and reporting them subsequently.

Cole’s mergesort [26] is a pipelined algorithm which works at several levels of the tree at once and overlaps the merging process at different nodes. Let us consider left sublist \(A_l = \{A_1, \ldots, A_{mid}\}\) and right sublist \(A_r = \{A_{mid+1}, \ldots, A_n\}\) of edge indices present in two children of an internal node of the binary merge tree whose leaf nodes contain the edge indices. The inversions in the leaf nodes have to be identified and counted while sorting \(A_l\) and \(A_r\). Let the number of inversions found in \(A_l\) and \(A_r\) be \(Inv_l\) and \(Inv_r\) respectively. To illustrate this point, let us consider that in a given timestep while merging the two sublists, we are at index \(i\) in list \(A_l\) and at index \(j\) in list \(A_r\). Since, \(A_l\) and \(A_r\) are already sorted, if at any step, \(A_l[i] > A_r[j]\), then \(A_l[i+1], \ldots, A_l[mid]\) will also be greater than \(A_r[j]\). As such \(Inv_m\) for an element at \(j\) consists of set \(\{(i, j), (i + 1, j), \ldots, (mid, j)\}\) containing \(mid - i + 1\) inversions. \(Inv_l\) and \(Inv_r\) are added to the number of inversions found during the merging \((Inv_m)\) of sorted list \(A_l\) and \(A_r\) for each internal node of the binary tree. With this modification, the total number of inversions \((Inv_{root})\) is available in the root node of the merging tree when mergesort algorithm terminates.

In the last timestep, remaining elements are compared. At each timestep, inversions are marked by utilizing the cross-rank of the elements computed by the original Cole’s algorithm in left and right sublists. Please refer to [26] for details on Cole’s algorithm.

**Identifying pairs of intersecting segments:** As described above, in the extended Cole’s mergesort algorithm, each non-leaf node has to update \(Inv_l\), \(Inv_r\), \(Inv_m\) and add them together to generate \(Inv\) for its parent. Based on the total count \(K\) of inversions found across all the scanbeams, \(K\) additional processors are allocated which can be \(O(n^2)\). These are subsequently allocated proportionately to individual scanbeams \((Inv_{root})\) and to the individual mergetree nodes for each of the \(O(\log n)\) merging steps of \((i, j)\) index pairs which have non-zero inversions based on their respective inversion counts. The merging process is repeated to enable recording of the inversions. In the process of merging, for each set of inversions found for the \(j\)th element from \(A_r\), the processors assigned to the node (as in Cole’s original algorithm) are utilized to copy the elements from location \(i\) to \(mid\) from \(A_l\) and \(j\) to two different arrays \(I\) and \(J\) respectively. Two additional auxiliary arrays \(Cnt\) and \(Sum\) are also obtained from the initial mergesort to aid this process. \(Cnt\) stores the \((mid - i + 1)\) values (for \(O(\log n)\) time steps, thus of total size \(O(n \log n)\)) and \(Sum\) stores the prefix sum of \(Cnt\). This computation for inversion counting and prefix sum can both be accomodated in the original run of the mergesort algorithm without sacrificing its \(O(\log n)\) time complexity using \(O(n)\) processors.

Using \(Cnt\) and \(Sum\) arrays, a processor reads one element from \(I\) and the other from \(J\) and in this way, \(Inv_{root}\) number of processors can report all the inversions, which represents indices of the intersecting pair of edges in a scanbeam, in constant time.

**Plane-Sweep Based Algorithm:** With these results, we discuss the parallelization of Vatti’s clipping algorithm. The parallel algorithm consists of four steps and is given next as Algorithm 1. The scanbeams are generated by Step 1. Step 2 is meant for populating the scanbeams with edges from input polygons in parallel. In subsection III-E we show the application of segment trees to accomplish the partitioning task in logarithmic time.

<table>
<thead>
<tr>
<th>Time Step(i)</th>
<th>Comparison ((A_l(i), A_r(i)))</th>
<th>Merged List</th>
<th>Inversions</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1,3</td>
<td>1,2,3,4,5,6,7,9</td>
<td>(7,1), (7,2), (7,4), (7,3), (5,3), (6,3), (9,3)</td>
</tr>
<tr>
<td>3</td>
<td>5,1</td>
<td>1,2,4,5,6,9</td>
<td>(5,1), (5,2), (5,4), (6,1), (9,1)</td>
</tr>
<tr>
<td>2</td>
<td>6,2</td>
<td>2,4,6,9</td>
<td>(6,2), (6,4), (9,2)</td>
</tr>
<tr>
<td>1</td>
<td>9,4</td>
<td>4,9</td>
<td>(9,4)</td>
</tr>
</tbody>
</table>

**TABLE I:** Example shows the merging of sublists \(A_l\) and \(A_r\) in an internal node of Cole’s mergesort tree in a time-stepped fashion. \(A_l = \{5, 6, 7, 9\}\) and \(A_r = \{1, 2, 3, 4\}\). Also note the inversions marked for reporting by our extended Cole’s merging algorithm.

<table>
<thead>
<tr>
<th>Scanbeam</th>
<th>Edges</th>
<th>Partial Polygon</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1 – s7</td>
<td>c1,c9,s1,c7,s1,s2,c2,c3</td>
<td>s1,s7</td>
</tr>
<tr>
<td>s7 – c9</td>
<td>c1,c9,s7,s6,s1,s2,c2,c3</td>
<td>s7,c1</td>
</tr>
<tr>
<td>c9 – c3</td>
<td>c9,c8,s7,s6,s1,s2,c2,c3</td>
<td>s9,c7</td>
</tr>
<tr>
<td>c3 – c4</td>
<td>c9,c4,c1,c4,s7,s6,s1,s2</td>
<td>s9,c3</td>
</tr>
<tr>
<td>s2 – s8</td>
<td>s7,s8,c9,c4,c4,c2,s2,s3</td>
<td>s2,s7,s8,s9</td>
</tr>
<tr>
<td>c5 – s3</td>
<td>c5,c6,c6,c6,c6,c6,c6,c6</td>
<td>s5,s6,s7,s8,s9</td>
</tr>
<tr>
<td>s3 – s5</td>
<td>c5,c6,c6,c6,c6,c6,c6,c6</td>
<td>s5,s6,s7,s8,s9</td>
</tr>
<tr>
<td>c4 – c6</td>
<td>c4,c7,c4,c7,c4,c7,c4,c6</td>
<td>s5,s6,s7,s8,s9</td>
</tr>
</tbody>
</table>

**TABLE II:** Scanbeam table showing the edges and labeled vertices of output polygons after processing intersections, start/end points for all scanbeams

Here, partitioning of polygon edges by \(m\) scanlines leads to
Lemma 1 Plane-Sweep based Divide-and-Conquer Algorithm

\textbf{INPUT:} \( V = V_b \cup V_o, \ E = E_b \cup E_o \) and clipping operator \( \text{op} \in \{ \cap, \cup, \setminus \} \)

\textbf{OUTPUT:} \( P \leftarrow B \text{ op } O \)

Step 1: Sort \( V \) by the y-coordinates.
Step 2: Partition \( E \) into \( m \) subsets \( E_1, E_2, \ldots, E_m \) of edges where \( E_i \) belongs to \( i \)th scanbeam and \( m \) is the number of scanbeams.
Step 3:
\begin{algorithmic}
  \FORALL {\( E_i \) in \( E \) in parallel do}
    \STATE Get y-coordinate of lower(\( y_b \)) and upper(\( y_t \)) scanline
    \STATE \text{Step 3.1:} \( P_i \leftarrow \text{Initialize}(y_b) \) \text{ // process minima}
    \STATE \text{Step 3.2:} \( P_i \leftarrow P_i \cup \text{EdgeIntersection}(y_b, y_t) \)
    \STATE \text{Step 3.3:} \( P_i \leftarrow P_i \cup \text{Terminate}(y_t) \) \text{ //process maxima}
    \STATE \text{Step 3.4:} Arrange the vertices in \( P_i \) in correct order
  \ENDFOR
\end{algorithmic}
Step 4: \( P \leftarrow (P_1 \cup P_2 \cup \ldots \cup P_m) \)

The intersection operation results in convex output since the trapezoids are themselves convex in nature. On the other hand union operation may result in concave output.

Fig. 5: Labeling of self-intersection \( I \) as left and right

Fig. 6: Merging partial output polygons from scanbeams. The arrows show the order of merging. \( P_1 \) and \( P_2 \) are the output polygons

For the intersection operation, the vertices with left and right labels are kept in two subgroups (bounds). The vertices with left label are sorted in ascending order by y-coordinate to form left bound and the vertices with right label are sorted in descending order by y-coordinate to form right bound. Then, the left and right bounds are concatenated to yield a partial polygon \( P_i \) based on the parity of the scanbeam id. For scanbeams with even parity, right bound is followed by left bound and for scanbeams with odd parity, left bound is followed by right bound. Figure 3 and 5 shows different scenarios of processing the labeled vertices in a scanbeam.

There may be single partial polygon as shown in Figure 3 (a, b and d) or multiple partial polygons as in Figure 3 (c and e) in a scanbeam. When there is self-intersection \( I \) as shown in Figure 5 in a scanbeam, it is duplicated and labeled as left and right. The partial polygon in this scenario is \((D_R, I_R, B_R, A_L, I_L, C_L)\). As we can see, the local merging is performed simply by concatenating bounds since the input partial polygons are convex in nature. Similary, for other clipping operations like union and difference, local merging can be done using simple concatenation of bounds...
as described in \cite{27}.

**Merging partial output polygons:** We partitioned the input polygons by introducing extra virtual vertices which makes merging by union operation straightforward since partial polygons $P_i$ and $P_{i+1}$ from two adjacent scanbeams may have a common horizontal edge or a vertex. If the partial polygons do not share a common edge/vertex, then union operation simply adds the partial polygon to the list of output polygons. Otherwise, the union operation can be implemented by concatenating the vertex list of polygons in a certain order as shown in Figure 5. Partial polygons in two consecutive scanbeams are concatenated together at a time and subsequently rearranged for merging in the next phase. The merging is described as $m$-leaf complete binary tree with partial polygons placed at the leaves of the tree and the internal nodes are formed by the union of the partial polygons corresponding to its two children. Thus, $O(\log m)$ phases are required with $m/2$ concurrent polygon union in the first phase and half the number of union in subsequent phases with respect to the previous phase thereby following a reduction tree approach. After each partial polygon union operation, vertices are re-ordered based on the parity of the scanbeam id. If the partial polygons share a common vertex, the merging can be performed in the same way as self-intersections are handled as shown in Figure 5. The virtual vertices are removed finally by array packing.

**E. Complexity Analysis**

The analysis is based on CREW PRAM model. The input for Step 1 is the start and end vertices of both input polygons. Assuming polygon $B$ and $O$ has $n$ vertices each, the sorting of $2n$ vertices can be performed using Cole's $O(\log n)$ time parallel merge sort algorithm \cite{26} using $O(n)$ processors. The input to Step 2 is the set of edges $E$ from both input polygons and the output is $m$ subsets of $E$ where $E_i$ represents edges in the $i$th scanbeam. Here, $m$ is equal to the number of intervals created by sorting the start and end vertices of edges of both input polygons by $y$ coordinate. Since there are $2n$ vertices, $m < 2n$. Partitioning $E$ into $m$ subsets requires intersection of $m$ scanlines with polygon $B$ and $O$ thereby producing $k'$ virtual vertices.

**Using segment tree for Step 2:** This step can be carried out in two phases using segment tree. The number of edges in the scanbeams need to be counted first. In the second phase, processors need to be allocated based on the count and these processors can report the edges in the scanbeams. To find the edges in a given scanbeam efficiently, a segment tree $T$ is constructed in parallel with a simple modification that each node stores count which is the size of the cover list $c$. With this modification, the number of edges in a scanbeam can be counted by simply traversing $T$ in $O(\log m)$ time without going through the edges in $c$. $T$ is queried concurrently by $m$ processors, each processor querying for $y_i \in \{(y_0 + y_1)/2, (y_1 + y_2)/2, \ldots, (y_{n-2} + y_{n-1})/2\}$ in $O(\log n)$ time while keeping track of the indices of the nodes visited and adding the count while visiting the nodes. It should be noted that the edges in $c$ are not reported at this stage, instead its count $|c|$ is read. Each query $q_i$ takes $y_i$ as input and returns $k_i$ where $k_i$ is a set of tuples $(k_{i1}, |c_{i1}|), (k_{i2}, |c_{i2}|), \ldots, (k_{ilogm}, |c_{ilogm}|)$, $0 \leq k_{ij} \leq 2^{|y_i-1|}$, where $i$ denotes scanbeam, $j$ denotes the level of the node and $k_{ij}$ denotes the index of the node in $T$. The number of edges spanning each scanbeam denoted by $k'_i$ is $(|c_{i1}| + |c_{i2}| + \ldots + |c_{ilogm}|)$. Now we can allocate $k'$ processors in total for Step 2 where $k'$ is $(k'_1 + k'_2 + \ldots + k'_m)$ for reporting edges in $c$. The number of processors that can be assigned to a node at $k_{ij}$ is given by $(|c_{i0}| + |c_{i1}| + \ldots + |c_{ilogm}|)$. Since $k_i$ contains the count of edges and their location in $T$, $k'$ processors can be distributed among the different nodes in $T$ and assigned to the edges in $c_{ij}$ for reporting edges in the scanbeams in $O(\log n)$ time ($m$ is $O(n)$ in the worst case). Since processor requirement is output-sensitive in nature, an $O(\log n)$ time overhead is incurred due to processor allocation.

The input to Step 3 is $O(n + k' + k)$ vertices which are divided among different scanbeams and the number of vertices to be processed in a scanbeam may vary from one scanbeam to another. Let $I$ denote the output size where $I \leq n + k' + k$. Step 3 is executed in parallel for all the scanbeams. Based on Lemma 2, vertices lying on top and bottom scanlines are checked if they are contributing vertices in Step 3.1 and 3.3. In Lemma 3, it is shown that a prefix sum operation is required to determine contributing vertices for edges of a polygon in a scanbeam in $O(\log n)$. In the worst case, there may be $O(n)$ edges in a scanbeam. As such, four prefix sum operations (two for lower scanline and two for upper scanline) are required for each scanbeam. Since we have allocated $O(n + k')$ processors a priori, this step takes $O(\log n)$ time. From Lemma 4, we can allocate $k$ processors in $O(\log n)$ time to report $O(k)$ intersecting vertices and label them in constant time in Step 3.2. Step 3.4 involves local merging of $O(I)$ vertices by sorting and array packing and can be done in $O(\log I)$ time using $O(I)$ processors. As such, Step 3 can be done in $O(\log I)$ time using $O(I)$ processors. Step 4 can also be done in logarithmic time since it involves sorting of the $m$ partial polygon list by their scanbeam id’s and list concatenation only.

Here, we discussed the intersection operation but the same analysis holds for union and difference as well because every intersection of two edges is a vertex of the final output polygon irrespective of the clipping operation performed. Moreover, contributing vertices can be determined using Lemma 3 for all clipping operations. Thus, the time complexity of Algorithm 1 dominated by Step 3, is $O((n + k + k')\log(n + k + k'))/p$ using $p \leq O(n + k' + k)$ processors. Since $k$ and $k'$ can be no more than $O(n^2)$, we get $O(\log n)$ time algorithm using $(n + k + k')$ processors.

**IV. Multi-threaded Multiway Divide and Conquer Algorithm**

In section III, we described PRAM algorithm. This section describes the multi-threaded implementation of Vatti’s algorithm. Algorithm 2 describes the algorithm for two input polygons, but also extends to a larger set of polygons. In short,
the inputs polygons are partitioned into multiple slabs, each slab is sequentially processed and finally, the partial output is merged together. The start and end vertices of the polygonal edges are sorted first. Then, Minimum Bounding Rectangle (MBR) of the union of input polygons is computed in Step 3. After the partitioning in Steps 4 and 5, partial polygons are sequentially clipped in \( p \) horizontal slabs in Step 6. It should be noted that any sequential clipping algorithm can be used in these steps. We employ General Polygon Clipper (GPC) \([28]\) for sequential polygon clipping in Step 6. GPC library is a robust polygon clipping library which takes as input two polygons or two sets of polygons and uses a variation of plane-sweep algorithm as described in \([29]\). However, in steps 4 and 5, we used Greiner-Hormann (GH) algorithm \([23]\) since we found it to be faster than GPC for rectangular clipping. Step 8 is currently sequentially done as it is a smaller fraction of the time, but can be parallelized as illustrated in Fig. 6 for stronger scaling.

In order to handle two sets of input polygons, at first, we form the event point list \( L \) for plane-sweep by adding the two \( Y \) coordinates of polynomial MBRs in \( L \) and sorting it in ascending order. We partition event points in \( L \) into \( p \) horizontal slabs where \( p \) is the number of threads such that every thread gets roughly equal number of local event points. This approach is different from statically segmenting the input polygons in a uniform grid, instead, it relies on the distribution of polygons by taking into account its MBR. Each thread determines the input polygons in its slab. Instead of splitting the polygons spanning consecutive slabs, we replicate those polygons in the slabs. The local event list is readjusted such that no polygon is partially contained in a given slab. Now, the merging phase is not required. However, there may be redundant output polygons which can be eliminated as a post-processing step. Finally, \( p \) number of plane-sweep based polygon clipping can be concurrently launched, where \( p \) is the number of threads.

V. EXPERIMENTAL SETUP AND IMPLEMENTATION RESULTS

To show the performance of multi-threaded algorithms, we present results that we obtained, on both simulated and real data. It contains two versions: one programmed using Java threads for two polygons and another using Pthreads for two sets of polygons. The experiments were performed on 64-bit CentOS operating system running on AMD Opteron (TM) Processor 6272 (1.4 GHz) with 64 CPU cores and 64 GB memory. The Pthread program was compiled using GCC 4.4.6 with O3 optimization enabled.

A. Synthetic Data

To measure the scalability of Algorithm 2 for a pair of polygons, we created a small test program to produce two polygons namely subject polygon and clip polygon with different number of edges. We generated input polygons with

---

Algorithm 2 Multi-threaded Polygon Clipping

**Input:** polygon \( A \) and \( B \) with vertices \( V_A \) and \( V_B \) clipping operator \( op \in \{\text{union, intersection, difference}\} \)

\( p \leftarrow \text{thread count}, \ id \leftarrow \text{threadId} \)

**Output:** polygon set \( C \)

1: add distinct \( y \)-coordinates of \( V_A \) and \( V_B \) to \( y \)
2: sort \( y \)
3: rectangle \( U \leftarrow \text{compute minimum bounding rectangle of } A \cup B \)
4: \( \text{for all slab in parallel do} \)
5: \( \text{rect denotes rectangular slab for a thread} \)
6: \( A_{id} \leftarrow \text{rectangleClip}(A, \text{rect}) \)
7: \( B_{id} \leftarrow \text{rectangleClip}(B, \text{rect}) \)
8: \( C_{id} \leftarrow \text{polygonClip}(A_{id}, B_{id}, op) \)
9: \( C \leftarrow (C_0 \cup C_1 \cup \ldots \cup C_{p-1}) \) // sequential

---

1. MBR is represented using bottom-left and top-right vertices with \( X \) and \( Y \) coordinates.

---

8
clipping subproblems and since since GPC library is relatively better at clipping smaller polygons in comparison to larger polygons as shown in Figure 7, we get better performance due to multi-threading and partitioning polygons. Practically, we found more than two fold speedup for larger polygons when number of threads is doubled from 1 to 2 and from 2 to 4.

In Figure 9, two sets of data (denoted by I and II in the figure) are used to find the execution time for partitioning (Step 4 and 5), clipping (Step 6) and merging (Step 8) as discussed in Algorithm 2. As expected, being the most complex operation, the clipping phase takes more time than partitioning and merging phases. Moreover, the average time taken for partitioning slightly increases as the number of threads increase.

<table>
<thead>
<tr>
<th>Number</th>
<th>Dataset</th>
<th>Polys</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ne_10m_urban_areas</td>
<td>11,878</td>
<td>1,153,348</td>
</tr>
<tr>
<td>2</td>
<td>ne_10m_states_provinces</td>
<td>4,647</td>
<td>1,332,830</td>
</tr>
<tr>
<td>3</td>
<td>GML_data_1</td>
<td>101,860</td>
<td>4,488,080</td>
</tr>
<tr>
<td>4</td>
<td>GML_data_2</td>
<td>128,682</td>
<td>6,262,858</td>
</tr>
</tbody>
</table>

**Table III: Description of real-world datasets**

**B. Real Data**

As real-world data, we selected polygonal data from Geographic Information System (GIS) domain. We experimented with shapefile available for free, from http://www.naturalearthdata.com. The details of the datasets are provided in Table III. The average edge length for the first dataset is 0.00415 with standard deviation of 0.0101. The average edge length for the second dataset is 0.0282 with standard deviation of 0.0546. We also experimented with GML (Geographic Markup Language) data related to telecommunication domain [31].

As we can see in Figure 10, intersection and union of larger datasets (3 and 4 in Table III) scales better than smaller datasets (1 and 2). To obtain absolute speedup, we performed clipping operation using ArcGIS which is state of the art software for GIS datasets. For the same datasets, we also used GPC library but found ArcGIS to be faster. Figure 12 shows the absolute speedup gained by our multi-threaded implementation. It took 110 seconds for Intersect (3,4), 135 seconds for Union (3,4) and 28 seconds for Intersect (1,2) operations by ArcGIS. With this baseline performance, we got about 30 fold speedup for Intersect (3,4) and 27 fold speedup for Union (3,4). However, Intersect (1,2) operation when executed sequentially using GPC library, turned out to be about 5 times slower when compared to ArcGIS. It should be noted that we are using GPC library without any preprocessing step. A filter and refine strategy may improve the performance of our multi-threaded algorithm. Intersect (1,2) operation scales well upto 16 threads but only 3.4 fold speedup is obtained. The limited scalability can also be explained by relative load imbalance among threads as shown in Figure 11.
VI. CONCLUSION

Although polygon clipping and related problems have been researched since late seventies, the best algorithm \cite{1} was not output sensitive and did not handle arbitrary polygons. Our PRAM algorithm is developed from the first principles and it shows the parallelization of a plane-sweep based algorithm relying on parallel primitives such as prefix sum and sorting. We tested our multi-threaded algorithms with real-world and synthetic datasets and achieved 30x speedup using a 64-core AMD Opteron Processor. We have discussed in detail about the intersection operation but it can be easily extended to other operations for example union, difference, etc., by simply changing the vertex labeling rules. Our PRAM algorithm based on segment tree is suitable to be implemented on General Purpose Graphics Processing Unit (GPGPU) where a larger number of threads can be simultaneously scheduled for polygon clipping.

VII. ACKNOWLEDGEMENT

This material is based upon work supported by the National Science Foundation under Grant No. CCF 1048200 and CNS 1205650 (Dr. Prasad).

REFERENCES


